sean goedecke

How I use LLMs as a staff engineer

Software engineers are deeply split on the subject of large language models. Many believe they're the most transformative technology to ever hit the industry. Others believe they're the latest in a long line of hype-only products: exciting to think about, but ultimately not useful to professionals trying to do serious work.

Personally, I feel like I get a lot of value from AI. I think many of the people who don't feel this way are "holding it wrong": i.e. they're not using language models in the most helpful ways. In this post, I'm going to list a bunch of ways I regularly use AI in my day-to-day as a staff engineer.

Writing production code

I use Copilot completions every time I write code[1]. Almost all the completions I accept are complete boilerplate (filling out function arguments or types, for instance). It's rare that I let Copilot produce business logic for me, but it does occasionally happen. In my areas of expertise (Ruby on Rails, for instance), I'm confident I can do better work than the LLM. It's just a (very good) autocomplete.

However, I'm not always working in my areas of expertise. I frequently find myself making small tactical changes in less-familiar areas (for instance, a Golang service or a C library). I know the syntax and have written personal projects in these languages, but I'm less confident about what's idiomatic. In these cases, I rely on Copilot more. Typically I'll use Copilot chat with the o1 model enabled, paste in my code, and ask directly "is this idiomatic C?"

Relying more on the LLM like this is risky, because I don't know what I'm missing. It basically lets me operate at a smart-intern baseline across the board. I have to also behave like a sensible intern, and make sure a subject-matter expert in the area reviews the change for me. But even with that caveat, I think it's very high-leverage to be able to make these kinds of tactical changes quickly.

Writing throwaway code

I am much more liberal with my use of LLMs when I'm writing code that will never see production. For instance, I recently did a block of research which required pulling chunks of public data from an API, classifying it, and approximating that classification with a series of quick regexes. All of this code was run on my laptop only, and I used LLMs to write basically all of it: the code to pull the data, the code to run a separate LLM to classify it, the code to tokenize it and measure token frequencies and score them, and so on.

LLMs excel at writing code that works that doesn't have to be maintained. Non-production code that's only run once (e.g. for research) is a perfect fit for this. I would say that my use of LLMs here meant I got this done 2x-4x faster than if I'd been unassisted.

Learning new domains

Probably the most useful thing I do with LLMs is use it as a tutor-on-demand for learning new domains. For instance, last weekend I learned the basics of Unity, relying heavily on ChatGPT-4o. The magic of learning with LLMs is that you can *ask questions*: not just "how does X work", but follow-up questions like "how does X relate to Y". Even more usefully, you can ask "is this right" questions. I often write up something I think I've learned and feed it back to the LLM, which points out where I'm right and where I'm still misunderstanding. I ask the LLM *a lot* of questions.

I take a lot of notes when I'm learning something new. Being able to just copy-paste all my notes in and get them reviewed by the LLM is great.

What about hallucinations? Honestly, since GPT-3.5, I haven't noticed ChatGPT or Claude doing a lot of hallucinating. Most of the areas I'm trying to learn about are very well-understood (just not by me), and in my experience that means the chance of a hallucination is pretty low. I've never run into a case where I learned something from a LLM that turned out to be fundamentally wrong or hallucinated.

Last resort bug fixes

I don't do this a lot, but sometimes when I'm really stuck on a bug, I'll attach the entire file or files to Copilot chat, paste the error message, and just ask "can you help?"

The reason I don't do this is that I think I'm currently much better at bug-hunting than current AI models. Almost all the time, Copilot (or Claude, for some personal projects) just gets confused. But it's still worth a try if I'm genuinely stuck, just in case, because it's so low-effort. I remember two or three cases where I'd just missed some subtle behaviour that the LLM caught, saving me a lot of time.

Because LLMs aren't that good at this yet, I don't spend a lot of time iterating or trying to un-stick the LLM. I just try once to see if it can get it.

Proofreading for typos and logic mistakes

I write a fair amount of English documents: ADRs, technical summaries, internal posts, and so on. I **never** allow the LLM to write these for me. Part of that is that I think I can write more clearly than current LLMs. Part of it is my general distaste for the ChatGPT house style.

What I do occasionally do is feed a draft into the LLM and ask for feedback. LLMs are great at catching typos, and will sometimes raise an interesting point that becomes an edit to my draft.

Like bugfixing, I don't iterate when I'm doing this - I just ask for one round of feedback. Usually the LLM offers some stylistic feedback, which I always ignore.

Summary

I use LLMs for these tasks:

- Smart autocomplete with Copilot
- Short tactical changes in areas I don't know well (always reviewed by a SME)
- Writing lots of use-once-and-throwaway research code
- Asking lots of questions to learn about new topics (e.g. the Unity game engine)
- Last-resort bugfixes, just in case it can figure it out immediately
- Big-picture proofreading for long-form English communication

I don't use LLMs for these tasks (yet):

- Writing whole PRs for me in areas I'm familiar with
- Writing ADRs or other technical communications
- Research in large codebases and finding out how things are done

1. Disclaimer: I work for GitHub, and for a year I worked directly on Copilot. Now I work on GitHub Models.

   ↩

February 4, 2025